Rewriting Logic[1]

Mark Hills
mhills@cs.uiuc.edu
University of Illinois at Urbana-Champaign

November 26, 2007

[1] Most material based on slides by José Meseguer

Equational Theories

Theories in equational logic are called *equational theories*, or *algebraic specifications*. A theory is a pair, (Σ, E):

- Σ is the signature, or syntax, of the theory
- E is a set of equations between expressions, or terms, over the syntax of Σ

Signatures

Signatures describe the syntax, and come in three varieties, based on the available *sorts*, or *types*, of the signature:

- *Unsorted*, or *single-sorted*, signatures, include only a single sort; all operations are over this one sort
- *Multi-sorted* signatures include many different, unrelated sorts
- *Order-sorted* signatures include many different sorts organized in a partial order based on inclusion; for instance, natural numbers are a subset of integers, so `Nat` $<$ `Integer`

Single-Sorted Example

```
fmod NAT-PREFIX is
  sort Natural .
  op 0 : -> Natural .
  op s : Natural -> Natural .
  op plus : Natural Natural -> Natural .
  vars N M : Natural .
  eq plus(N,0) = N .
  eq plus(N,s(M)) = s(plus(N,M)) .
endfm
```

Note: Operations with arity 0 are *constants*.

Single-Sorted Example, with Mix-fix

```
fmod NAT-MIXFIX is
  sort Natural .
  op 0 : -> Natural .
  op s_ : Natural -> Natural .
  op _+_ : Natural Natural -> Natural .
  op _*_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s M = s(N + M) .
  eq N * 0 = 0 .
  eq N * s M = N + (N * M) .
endfm
```

Many-Sorted Signatures

Many-sorted signatures are made up of operations over a set of sorts S. A many-sorted signature can be defined mathematically as:

$$\Sigma = (S, \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S})$$

where w is a sequence of 0 or more sorts ($\in S^*$). An operation $f \in \Sigma_{w,s}$ is denoted as:

$$f : w \rightarrow s$$

Many-Sorted Example

```
1  fmod NAT-LIST is
2    protecting NAT-MIXFIX .
3    sort List .
4    op nil : -> List .
5    op _._ : Natural List -> List .
6    op length : List -> Natural .
7    var N : Natural .
8    var L : List .
9    eq length(nil) = 0 .
10   eq length(N . L) = s length(L) .
11 endfm
```


Many-Sorted Example, Signature

```
1 sorts Natural List .
2 op 0 : -> Natural .
3 op s_ : Natural -> Natural .
4 op _+_ : Natural Natural -> Natural .
5 op _*_ : Natural Natural -> Natural .
6 op nil : -> List .
7 op _._ : Natural List -> List .
8 op length : List -> Natural .
```


Many-Sorted Example, Mathematical Signature

- $S = \{\texttt{Natural}, \texttt{List}\}$
- $\Sigma_{nil,\texttt{Natural}} = \{0\}$
- $\Sigma_{nil,\texttt{List}} = \{\texttt{nil}\}$
- $\Sigma_{\texttt{Natural},\texttt{Natural}} = \{\texttt{s}\_\}$
- $\Sigma_{\texttt{Natural Natural},\texttt{Natural}} = \{\_ + \_, \_ * \_\}$
- $\Sigma_{\texttt{Natural List},\texttt{List}} = \{\_.\_\}$
- $\Sigma_{\texttt{List},\texttt{Natural}} = \{\texttt{length}\}$
- all other $\Sigma_{w,s} = \emptyset$

Order-Sorted Signatures

Order-sorting allows natural ways of organizing sorts to deal with common definitional problems:

- partial operations: operations may be partial, but may be total on some other sort

```
1    op succ_ : Natural -> NzNatural .
2    op _._   : Natural List -> NeList .
3    op first : NeList -> Natural .
4    op pred_ : NzNatural -> Natural .
```

- overloading: order-sorting allows overloading of operations, with natural meanings

```
1    op _+_ : Natural Natural -> Natural .
2    op _+_ : Integer Integer -> Integer .
```


Order-Sorted Signatures, Mathematically

An order-sorted signature Σ is a triple

$$\Sigma = (S, \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S}, <)$$

where $(S, \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S})$ is a multi-sorted signature (also called an S-sorted signature, since we have sorts S) and $<$ is a partial order relation on S called *subsort inclusion*.

Kinds and Connected Components

Sorts in an order-sorted signature form equivalence classes based on the order-sorting relationship.

- Graph of sorts partitioned into connected components
- Each connected component is an equivalence class
- Will often hear these referred to as *kinds*; can think of kinds as the name of the connected component (and, in Maude, if you get a result with a kind, not a sort, it probably means you did something wrong...)

Algebras

Given a signature like $\Sigma = (S, \{\Sigma_{w,s}\}_{(w,s) \in S^* \times S})$, we need a way to assign meaning to it. Meanings are given in specific *models*, called Σ-algebras, defined by:

- an S-indexed family of sets $A = \{A_s\}_{s \in S}$
- for each $a : nil \rightarrow s$ in Σ, an element $a_A^{nil,s} \in A_s$
- for each $f : w \rightarrow s$ in Σ, with $w = s_1 ... s_n, n > 0$, a function $f_A^{w,s} : A_{s_1} \times ... \times A_{s_n} \rightarrow A_s$

Term Algebras

One of the most common algebras to refer to is the *term algebra*, written T_Σ, made up of all the terms which can be formed over the signature Σ. In programming languages terms, we can think of this algebra as being made up of all possible, syntactically well-formed programs in the language.

Junk and Confusion

If term algebras are *sensible* (a technical definition not presented here, but basically meaning that overloaded operations yield the same results on the same inputs), they have two nice properties:

- Different terms denote different things (and, with this, the same term denotes *one* thing): no confusion
- The algebra is *minimal*, containing no terms it does not need: no junk

Evaluation

If term algebras contain, in syntax form, all the programs in a language, how can we run them? We use a *homomorphism* between the term algebra and an algebra representing the actual meaning of the language; this homomorphism maps terms in the term algebra to meanings in the language semantics.

- Homomorphisms preserve constants: each constant in the term algebra maps to a constant in the semantics.
- Homomorphisms preserve operations: each operation in the term algebra maps to an operation with the same arity in the semantics, with each argument to the operation mapped by the homomorphism as well.

Initiality

One question that may arise is: does an evaluation homomorphism from the term algebra T_Σ into an algebra A representing the semantics always exist? It does, under *initiality*:

If Σ is sensible, then for any Σ-algebra A there is a unique Σ-homomorphism $eval_A : T_\Sigma \rightarrow A$, an evaluation function that maps terms to their evaluated result in A.

Equations

Equations provide a method of identifying equal terms.

- Equational logic comes with a sound, complete deduction system based on equational reasoning
- Terms which are provably equal are grouped into equivalence classes, modulo the given equations; all terms in an equivalence class are in some sense identical
- Equations applied to either entire term or subterms using matching; a proper substitution θ must be found from each variable in the equation to subterms of the matched term

Why Rewriting Logic

In equational logic, terms are identified as equal based on equations using deduction. What about cases where terms are not equal?

- Concurrency: data races can lead to non-equivalent results
- Nondeterminism: nondeterministic choices can cause different execution paths to be taken

Equational reasoning is limited to deterministic choices – we need a way to represent non-deterministic choices.

Rewrite Theories

A rewrite theory $\mathcal{R}$ is a triple $\mathcal{R} = (\Sigma, E, R)$ where (Σ, E) is an equational theory and R is a set of *labeled rules*, each of the form $l : t \longrightarrow t'$, with l a label and $t, t' \in T_\Sigma$.

Example Rewrite Module

```
1  mod CANDY-AUTOMATON is
2    sort State .
3    ops $ ready broken nestle m&m q : -> State .
4    rl [in] : $ => ready .
5    rl [cancel] : ready => $ .
6    rl [1] : ready => nestle .
7    rl [2] : ready => m&m .
8    rl [fault] : ready => broken .
9    rl [chng] : nestle => q .
10   rl [chng] : m&m => q .
11 endm
```


Example Rewrite Module 2

```
1  mod PETRI-MACHINE is
2    sort Marking .
3    ops null $ c a q : -> Marking .
4    op _ _ : Marking Marking -> Marking [assoc comm id: null] .
5    rl [buy-c] : $ => c .
6    rl [buy-a] : $ => a q .
7    rl [chng] : q q q q => $ .
8  endm
```


Concurrency in Rewriting Logic

The above two examples showed both nondeterminism and concurrency. Rewriting logic allows representations of concurrency in subterms, at the top level, and in combinations, allowing both interleaved and "true" concurrency (modelling when two events actually happen at the same time, instead of modelling events as a series of sequentially interleaved steps).

Rewriting Logic and Programming Languages

The use of rewriting logic to define languages is referred to as *rewriting logic semantics*, and is an active area of study both here at UIUC and elsewhere. How are languages modelled?

- Syntax for languages is defined as operators
- A configuration is created for language evaluation, including the active computation and supporting infrastructure (memory, etc)
- Equations are used to define sequential steps in a computation
- Rules are used to define nondeterministic or concurrent steps, such as lock acquisition or memory operations

Executing Programs

To actually execute programs, it is necessary to be able to "run" the rules. This is done by converting equations and rules into term rewriting rules:

- Equations of the form $l = r$ are converted to rewrites of the form $l \rightarrow r$
- Rules of the form $l \Rightarrow r$ are converted to rewrites of the form $l \rightarrow r$

Note that operationally equations and rules are treated the same, although logically they are quite different – rules have a built-in direction, for instance, while in theory equations can apply both ways.

The KOOL Language

Switching to RTA presentation...

Material Not Covered

Much detail was left out to remain concise. This includes some details on Maude and the logic that it supports. More topics to look at include:

- Conditional equations and rules
- Memberships
- Sensible signatures
- Variables
- More relationships between algebras; homomorphisms and isomorphisms

To Learn More

If you find this material interesting (and I hope you do!), you should:

- Take CS476 with José Meseguer
- Take CS422 with Grigore Roşu
- Take a look at `http://maude.cs.uiuc.edu` and `http://fsl.cs.uiuc.edu`